Softwaretechnik / Software-Engineering

Lecture 12: Structural Software Modelling

2016-06-20

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Architecture & Design: Content

VL 11
- **Introduction and Vocabulary**
- **Principles of Design**
  - (i) modularity
  - (ii) separation of concerns
  - (iii) information hiding and data encapsulation
  - (iv) abstract data types, object orientation

⋮

- **Software Modelling**
  - (i) views and viewpoints, the 4+1 view
  - (ii) model-driven/-based software engineering
  - (iii) Unified Modelling Language (UML)
  - VL 12 (iv) **modelling structure**
    - a) (simplified) class diagrams
    - b) (simplified) object diagrams
    - c) (simplified) object constraint logic (OCL)

⋮

  - VL 13 (v) **modelling behaviour**
    - a) communicating finite automata
    - b) Uppaal query language
    - c) basic state-machines
    - VL 14 d) an outlook on hierarchical state-machines

⋮

- **Design Patterns**

⋮

… so, off to "'technological paradise' where […] everything happens according to the blueprints".

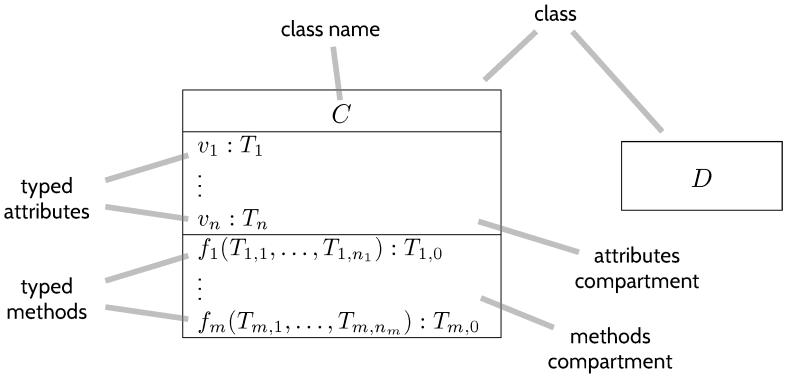
(Kopetz, 2011; Lovins and Lovins, 2001)

Content

- **Class Diagrams**
    - concrete syntax,
    - abstract syntax,
    - class diagrams at work,
    - semantics: system states.
- **Object Diagrams**
    - concrete syntax,
    - dangling references,
    - partial vs. complete,
    - object diagrams at work.
- **Proto-OCL**
    - syntax,
    - semantics,
    - Proto-OCL vs. OCL.
- Putting it All Together: **Proto-OCL vs. Software**

Class Diagrams

Class Diagrams: Concrete Syntax



class name

class

C

$v_1 : T_1$

$\vdots$

$v_n : T_n$

$f_1(T_{1,1}, \dots, T_{1,n_1}) : T_{1,0}$

$\vdots$

$f_m(T_{m,1}, \dots, T_{m,n_m}) : T_{m,0}$

D

typed attributes

typed methods

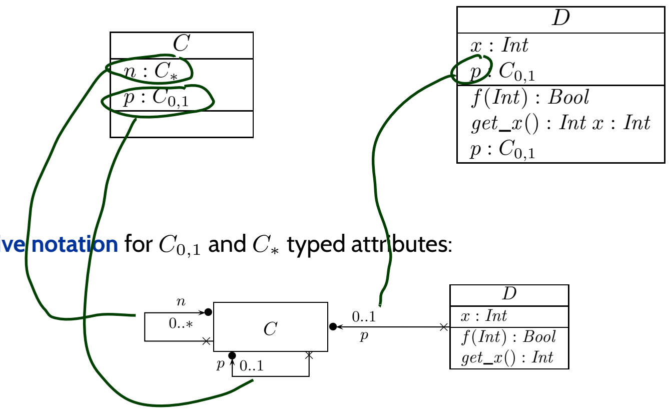
attributes compartment

methods compartment

where

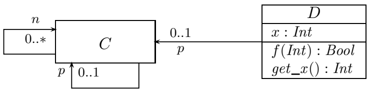
- $T_1, \dots, T_{m,0} \in \mathscr{T} \cup \{C_{0,1}, C_* \mid C$ a class name$\}$
- $\mathscr{T}$ is a set of **basic types**, e.g. $Int, Bool, \dots$.

Concrete Syntax: Example



Alternative notation for $C_{0,1}$ and C_* typed attributes:

Alternative lazy notation for **alternative notation**:



And nothing else! This is **the** concrete syntax of **class diagrams** for the **scope of the course**.

Abstract Syntax: Object System Signature

Definition. An **(Object System) Signature** is a 6-tuple

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$$

where

- $\mathscr{T}$ is a set of (basic) **types**,
- $\mathscr{C}$ is a finite set of **classes**,
- V is a finite set of **typed attributes** $v : T$, i.e., each $v \in V$ has type T,
- $atr : \mathscr{C} \to 2^V$ maps each class to its set of attributes.

powerset of V

- F is a finite set of **typed behavioural features** $f : T_1, \ldots, T_n \to T$,
- $mth : \mathscr{C} \to 2^F$ maps each class to its set of behavioural features.

we will discuss these not so much

- A type can be a basic type $\tau \in \mathscr{T}$, or $C_{0,1}$, or C_*, where $C \in \mathscr{C}$.

Note: Inspired by OCL 2.0 standard OMG (2006), Annex A.

Object System Signature Example

Definition. An (Object System) Signature is a 6-tuple

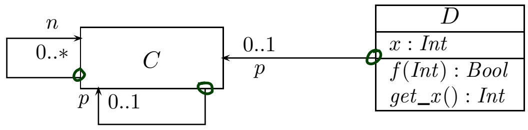
$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$$

where

- $\mathscr{T}$ is a set of (basic) types,
- $\mathscr{C}$ is a finite set of classes,
- V is a finite set of typed attributes $v : T$, i.e., each $v \in V$ has type T,
- $atr : \mathscr{C} \to 2^V$ maps each class to its set of attributes.
- F is a finite set of typed behavioural features $f : T_1, \dots, T_n \to T$,
- $mth : \mathscr{C} \to 2^F$ maps each class to its set of behavioural features.
- A type can be a basic type $\tau \in \mathscr{T}$, or $C_{0,1}$, or C_*, where $C \in \mathscr{C}$.

$$\begin{aligned}
\mathscr{S}_0 = (&\{Int, Bool\}, && \mathscr{T}\\
&\{C, D\}, && \mathscr{C}\\
&\{\underbrace{x : Int}, \underbrace{p : C_{0,1}}, \underbrace{n : C_*}\}, && V\\
&\{\underbrace{C \mapsto \{p, n\}}, \underbrace{D \mapsto \{p, x\}}\}, && atr\\
&\{\underbrace{f : Int \to Bool}, \underbrace{get\_x : Int}\}, && F\\
&\{\underbrace{C \mapsto \emptyset}, \underbrace{D \mapsto \{f, get\_x\}}\}) && mth
\end{aligned}$$

From Abstract to Concrete Syntax



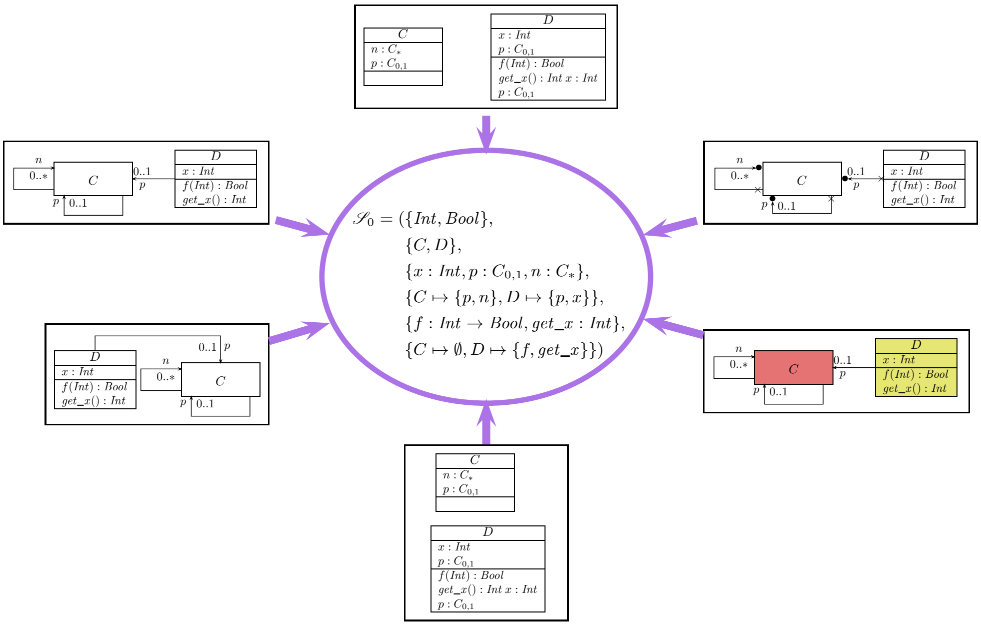
$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$$

- $\mathscr{T} = \{Int, Bool\}$,
- $\mathscr{C} = \{C, D\}$,
- $V = \{x : Int, p : C_{0,1}, n : C_*\}$,
- $atr = \{C \mapsto \{p, n\}, D \mapsto \{x, p\}\}$,
- $F = \{f : Int \to Bool, get\_x : Int\}$,
- $\underbrace{mth = \{C \mapsto \emptyset, D \mapsto \{f, get\_x\}\}}$

$mth(C) = \emptyset$
$mth(D) = \{f, get\_x\}$

$m : T_1, \dots, T_n \to T, \; n \geq 0$
$m : \to T$ if $n = 0$
$m : T$ if $n = 0$ also ok

Once Again: Concrete vs. Abstract Syntax

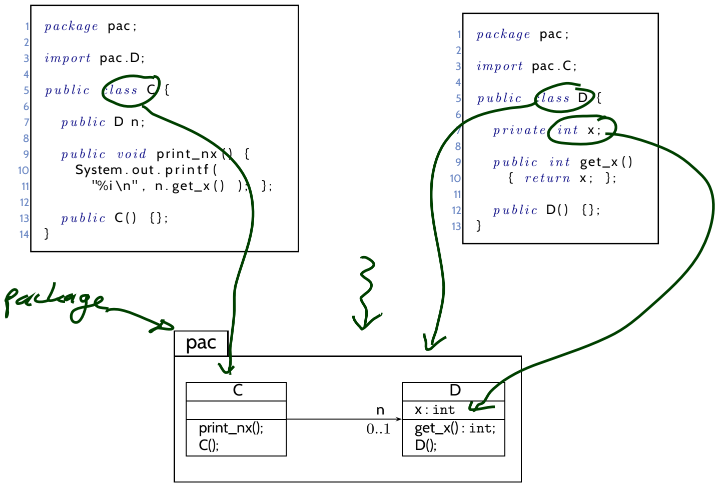


$$\begin{aligned}\mathscr{S}_0 = (&\{Int, Bool\},\\ &\{C, D\},\\ &\{x : Int, p : C_{0,1}, n : C_*\},\\ &\{C \mapsto \{p, n\}, D \mapsto \{p, x\}\},\\ &\{f : Int \to Bool, get\_x : Int\},\\ &\{C \mapsto \emptyset, D \mapsto \{f, get\_x\}\})\end{aligned}$$

Class Diagrams at Work

Visualisation of Implementation

- The class diagram syntax can be used to **visualise code**:
  provide rules which map (parts of) the code to class diagram elements.

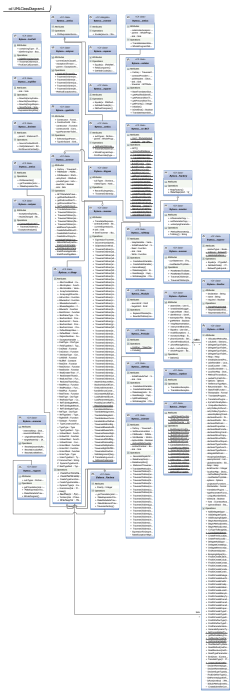


```
package pac;

import pac.D;

public class C {

  public D n;

  public void print_nx() {
    System.out.printf(
      "%i\n", n.get_x() ); };

  public C() {};
}
```

```
package pac;

import pac.C;

public class D {

  private int x;

  public int get_x()
    { return x; };

  public D() {};
}
```

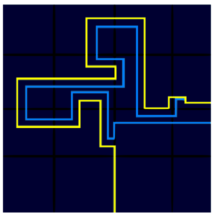
Visualisation of Implementation: (Useless) Example

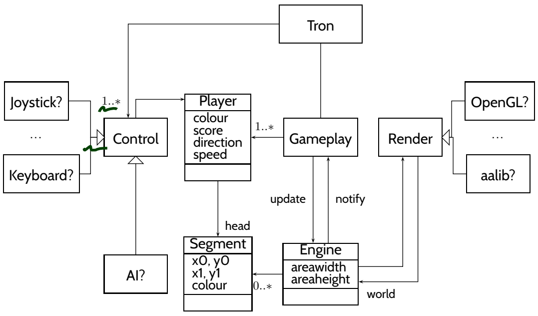
- open favourite IDE,
- open favourite **project**,
- press "**generate class diagram**"
- **wait**…**wait**…**wait**…



- ca. 35 classes,
- ca. 5,000 LOC C#

Visualisation of Implementation: (Useful) Example



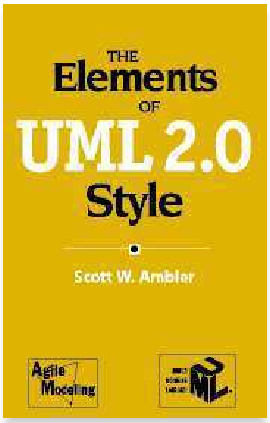


- **Note**: a class **diagram** for visualisation may be partial.

  → show only the **most relevant** classes and attributes (for the given purpose).
- **Note**: a signature can be defined by **a set of** class diagrams.

  → use multiple class diagrams with **a manageable** number of classes for different purposes.
- A diagram is **a good diagram** if (and only if?) it serves its **purpose**!

Literature Recommendation



(Ambler, 2005)

A More Abstract Class Diagram Semantics

Object System Structure

Definition. A Object System **Structure** of signature

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$$

is a domain function $\mathscr{D}$ which assigns to each type a domain, i.e.

- $\tau \in \mathscr{T}$ is mapped to $\mathscr{D}(\tau)$,
- $C \in \mathscr{C}$ is mapped to an infinite set $\mathscr{D}(C)$ of **(object) identities**.
  - object identities of different classes are disjoint, i.e. $\forall C, D \in \mathscr{C} : C \neq D \rightarrow \mathscr{D}(C) \cap \mathscr{D}(D) = \emptyset$,
  - on object identities, (only) comparision for equality "$=$" is defined.
- C_* **and** $C_{0,1}$ for $C \in \mathscr{C}$ are mapped to $2^{\mathscr{D}(C)}$.

We use $\mathscr{D}(\mathscr{C})$ to denote $\bigcup_{C \in \mathscr{C}} \mathscr{D}(C)$; analogously $\mathscr{D}(\mathscr{C}_*)$.

Note: We identify **objects** and **object identities**,
because both uniquely determine each other (cf. OCL 2.0 standard).

Basic Object System Structure Example

Wanted: a structure for signature

$$\mathscr{S}_0 = (\{Int, \overset{\text{Flower}}{Bool}\}, \{C, D\}, \{\overset{y:\,\text{Flower}}{x : Int}, p : C_{0,1}, n : C_*\}, \{C \mapsto \{\overset{y}{p}, n\}, D \mapsto \{p, x\}\},$$
$$\{f : Int \to Bool, get\_x : Int\}, \{C \mapsto \emptyset, D \mapsto \{f, get\_x\}\})$$

A structure $\mathscr{D}$ maps

- $\tau \in \mathscr{T}$ to **some** $\mathscr{D}(\tau)$, $C \in \mathscr{C}$ to **some** identities $\mathscr{D}(C)$ (infinite, pairwise disjoint),
- C_* and $C_{0,1}$ for $C \in \mathscr{C}$ to $\mathscr{D}(C_{0,1}) = \mathscr{D}(C_*) = 2^{\mathscr{D}(C)}$.

$$\mathcal{D}(\text{Flower}) = \{\text{rose}, \text{daisy}, \text{lily}\}$$

$$\begin{aligned}
\mathscr{D}(Int) &= \mathbb{Z} \\
\mathscr{D}(C) &= \mathbb{N}^+ \times \{C\} = \{1_C, 2_C, 3_C, \dots\} \\
\mathscr{D}(D) &= \mathbb{N}^+ \times \{D\} = \{1_D, 2_D, 3_D, \dots\} \\
\mathscr{D}(C_{0,1}) = \mathscr{D}(C_*) &= 2^{\mathscr{D}(C)} \\
\mathscr{D}(D_{0,1}) = \mathscr{D}(D_*) &= 2^{\mathscr{D}(D)}
\end{aligned}$$

System State

Definition. Let $\mathscr{D}$ be a structure of $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth)$.
A **system state** of $\mathscr{S}$ wrt. $\mathscr{D}$ is a **type-consistent** mapping

$$\sigma : \underbrace{\mathscr{D}(\mathscr{C})}_{\text{object identities}} \overset{\text{partial function}}{\nrightarrow} (\overset{\text{attribute}}{V} \nrightarrow \overset{\text{value}}{(\mathscr{D}(\mathscr{T}) \cup \mathscr{D}(\mathscr{C}_*))}).$$

That is, for each $u \in \mathscr{D}(C)$, $C \in \mathscr{C}$, if $u \in \text{dom}(\sigma)$

- $\text{dom}(\sigma(u)) = atr(C)$
- $\sigma(u)(v) \in \mathscr{D}(\tau)$ if $v : \tau, \tau \in \mathscr{T}$
- $\sigma(u)(v) \in \mathscr{D}(D_*)$ if $v : D_{0,1}$ or $v : D_*$ with $D \in \mathscr{C}$

We call $u \in \mathscr{D}(\mathscr{C})$ **alive** in σ if and only if $u \in \text{dom}(\sigma)$.

We use $\Sigma_{\mathscr{S}}^{\mathscr{D}}$ to denote the set of all system states of $\mathscr{S}$ wrt. $\mathscr{D}$.

System State Examples

$$\mathscr{S}_0 = (\{Int, \overset{Flower}{Bool}\}, \{C, D\}, \{x : Int, p : C_{0,1}, \overset{y : Flower}{n : C_*}\}, \{C \mapsto \{p, \overset{y}{n}\}, D \mapsto \{p, x\}\},$$

$$\{f : Int \to Bool, get\_x : Int\}, \{C \mapsto \emptyset, D \mapsto \{f, get\_x\}\})$$

$$\mathscr{D}(Int) = \mathbb{Z}, \quad \mathscr{D}(C) = \{1_C, 2_C, 3_C, ...\}, \quad \mathscr{D}(D) = \{1_D, 2_D, 3_D, ...\}$$

$\mathscr{D}(Flower) = \{rose, daisy, lily\}$

A system state is a partial function $\sigma : \mathscr{D}(\mathscr{C}) \nrightarrow (V \nrightarrow (\mathscr{D}(\mathscr{T}) \cup \mathscr{D}(\mathscr{C}_*)))$ such that

- $\mathrm{dom}(\sigma(u)) = atr(C)$,
- $\sigma(u)(v) \in \mathscr{D}(\tau)$ if $v : \tau, \tau \in \mathscr{T}$,
- $\sigma(u)(v) \in \mathscr{D}(C_*)$ if $v : D_*$ or $v : D_{0,1}$ with $D \in \mathscr{C}$.

$\sigma_1 = \emptyset$

$$\sigma_2 = \{1_C \mapsto \{y \mapsto rose, p \mapsto \{5_C\}, n \mapsto \emptyset\},$$
$$5_C \mapsto \{y \mapsto lily, p \mapsto \emptyset, n \mapsto \{1_C\}\},$$
$$4_D \mapsto \{x \mapsto 27, p \mapsto \{5_C\}\}\}$$

Content

- **Class Diagrams**
  - concrete syntax,
  - abstract syntax,
  - class diagrams at work,
  - semantics: system states.
- **Object Diagrams**
  - concrete syntax,
  - dangling references,
  - partial vs. complete,
  - object diagrams at work.
- **Proto-OCL**
  - syntax,
  - semantics,
  - Proto-OCL vs. OCL.
- Putting it All Together:
  Proto-OCL vs. Software

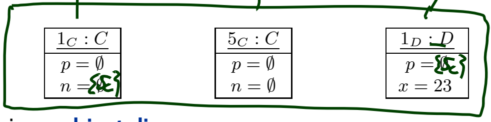
Object Diagrams

Object Diagrams

$$\mathscr{S}_0 = (\{Int, Bool\}, \{C, D\}, \{x : Int, p : C_{0,1}, n : C_*\}, \{C \mapsto \{p, n\}, D \mapsto \{p, x\}\},$$
$$\{f : Int \to Bool, get\_x : Int\}, \{C \mapsto \emptyset, D \mapsto \{f, get\_x\}\}), \qquad \mathscr{D}(Int) = \mathbb{Z}$$

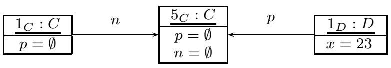
$$\sigma = \{\underbrace{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}}, \underbrace{5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}}, \underbrace{1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}}\}$$

- We may **represent** σ graphically as follows:

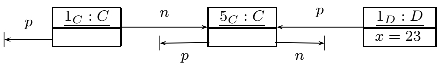


This is an **object diagram**.

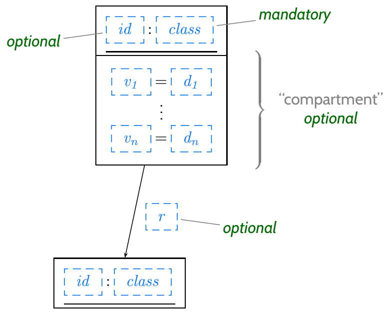
- Alternative notation:



- Alternative **non-standard** notation:



Concrete Syntax:



Special Case: Dangling Reference

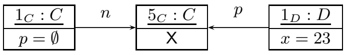
Definition.
Let $\sigma \in \Sigma_{\mathscr{S}}^{\mathscr{D}}$ be a system state and $u \in \mathrm{dom}(\sigma)$ an alive object of class C in σ.

We say $r \in atr(C)$ **is a dangling reference in** u if and only if $r : C_{0,1}$ or $r : C_*$ and u refers to a **non-alive** object via v, i.e.

$$\sigma(u)(r) \not\subseteq \mathrm{dom}(\sigma).$$

Example:

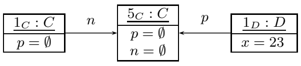
- $\sigma = \{\underbrace{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}}, \underbrace{1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}}\}$ $5_C \notin \mathrm{dom}(\sigma)$
- Object diagram representation:



Partial vs. Complete Object Diagrams

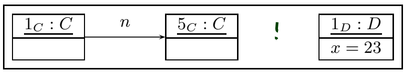
- By now we discussed "**object diagram represents system state**":

$$\begin{array}{l} \{1_C \mapsto \{p \mapsto \emptyset, n \mapsto \{5_C\}\}, \\ \;5_C \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, \\ \;1_D \mapsto \{p \mapsto \{5_C\}, x \mapsto 23\}\} \end{array} \quad \rightsquigarrow$$

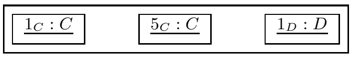


  What about the other way round…?

- **Object diagrams** can be **partial**, e.g.

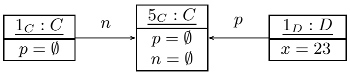


or



  → we may omit information.

- Is the following object diagram **partial** or **complete**? (wrt. given signature $\mathscr{S}$)

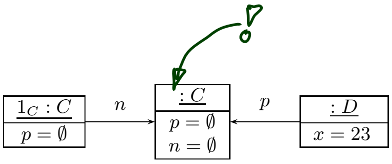


- If an object diagram
  - has values for **all** attributes of **all** objects in the diagram, and
  - if we **say that** it is meant to be complete

  then we can **uniquely** reconstruct a system state σ.

Special Case: Anonymous Objects

If the object diagram



is considered as **complete**, then it denotes the set of all system states

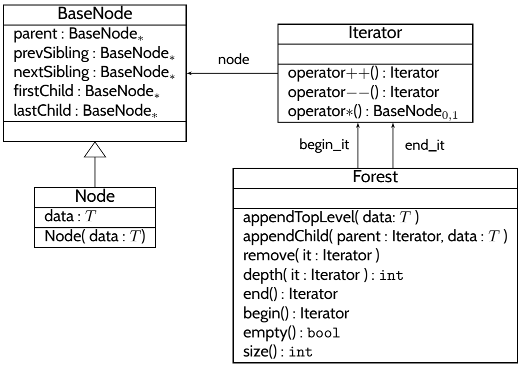
$$\{c_1 \mapsto \{p \mapsto \emptyset, n \mapsto \{c_2\}\}, c_2 \mapsto \{p \mapsto \emptyset, n \mapsto \emptyset\}, d \mapsto \{p \mapsto \{c_2\}, x \mapsto 23\}\}$$

where $\quad c \in \mathscr{D}(C), \quad d \in \mathscr{D}(D), \quad c \neq 1_C.$

Intuition: different boxes represent different objects.

Object Diagrams at Work

Example: Data Structure *(Schumann et al., 2008)*



BaseNode
parent : BaseNode$_*$
prevSibling : BaseNode$_*$
nextSibling : BaseNode$_*$
firstChild : BaseNode$_*$
lastChild : BaseNode$_*$

node

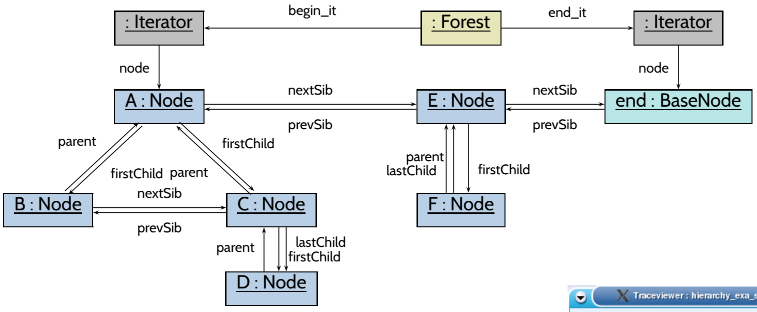
Iterator
operator$++$() : Iterator
operator$--$() : Iterator
operator$*$() : BaseNode$_{0,1}$

begin_it
end_it

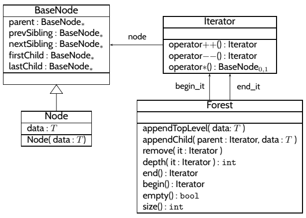
Node
data : T
Node(data : T)

Forest
appendTopLevel(data: T)
appendChild(parent : Iterator, data : T)
remove(it : Iterator)
depth(it : Iterator) : `int`
end() : Iterator
begin() : Iterator
empty() : `bool`
size() : `int`

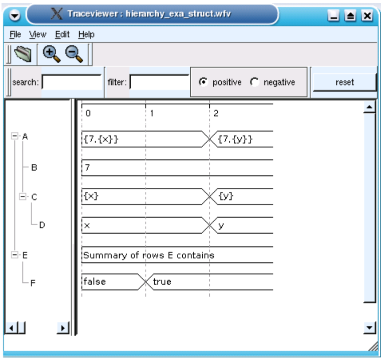
Example: Illustrative Object Diagram *(Schumann et al., 2008)*



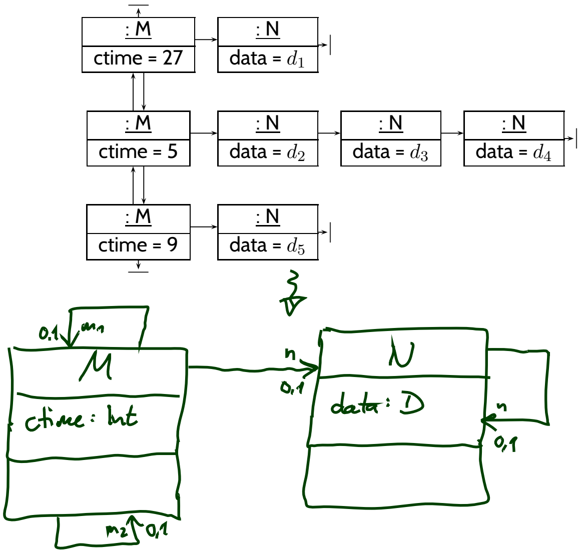
: Iterator
begin_it
: Forest
end_it
: Iterator
node
node
A : Node
nextSib
prevSib
E : Node
nextSib
prevSib
end : BaseNode
parent
firstChild
firstChild
parent
parent
lastChild
firstChild
B : Node
nextSib
prevSib
C : Node
F : Node
parent
lastChild
firstChild
D : Node



BaseNode
parent : BaseNode$_*$
prevSibling : BaseNode$_*$
nextSibling : BaseNode$_*$
firstChild : BaseNode$_*$
lastChild : BaseNode$_*$
node
Iterator
operator$++$() : Iterator
operator$--$() : Iterator
operator$*$() : BaseNode$_{0,1}$
begin_it
end_it
Node
data : T
Node(data : T)
Forest
appendTopLevel(data: T)
appendChild(parent : Iterator, data : T)
remove(it : Iterator)
depth(it : Iterator) : `int`
end() : Iterator
begin() : Iterator
empty() : `bool`
size() : `int`



Object Diagrams for Analysis

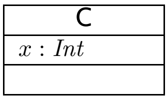


Content

- **Class Diagrams**
  - concrete syntax,
  - abstract syntax,
  - class diagrams at work,
  - semantics: system states.
- **Object Diagrams**
  - concrete syntax,
  - dangling references,
  - partial vs. complete,
  - object diagrams at work.
- **Proto-OCL**
  - syntax,
  - semantics,
  - Proto-OCL vs. OCL.
- Putting it All Together:
  Proto-OCL vs. Software

Towards Object Constraint Logic (OCL)
— “Proto-OCL” —

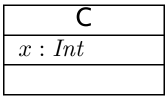
Constraints on System States



- **Example**: for all C-instances, x should never have the value 27.

$$\forall\, c \in allInstances_C \bullet x(c) \neq 27$$

Constraints on System States



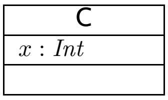
- **Example**: for all C-instances, x should never have the value 27.

$$\forall\, c \in allInstances_C \bullet x(c) \neq 27$$

- **Proto-OCL Syntax** wrt. signature $(\mathscr{T}, \mathscr{C}, V, atr, F, mth)$, c is a logical variable, $C \in \mathscr{C}$:

$$\begin{array}{rlll}
F ::= & c & : \tau_C & \\
| & allInstances_C & : 2^{\tau_C}, \; C \in \mathscr{C} & \\
| & v(F) & : \tau_C \to \tau_\perp, & \text{if } v : \tau \in atr(C), \; \tau \in \mathscr{T} \\
| & v(F) & : \tau_C \to \tau_D, & \text{if } v : D_{0,1} \in atr(C) \\
| & v(F) & : \tau_C \to 2^{\tau_D}, & \text{if } v : D_* \in atr(C) \\
| & f(F_1, \dots, F_n) & : \tau_1 \times \dots \times \tau_n \to \tau, & \text{if } f : \tau_1 \times \dots \times \tau_n \to \tau \\
| & \forall\, c \in F_1 \bullet F_2 & : \tau_C \times 2^{\tau_C} \times \mathbb{B}_\perp \to \mathbb{B}_\perp &
\end{array}$$

Constraints on System States



- **Example**: for all C-instances, x should never have the value 27.

$$\forall\, c \in allInstances_C \bullet x(c) \neq 27$$

- **Proto-OCL Syntax** wrt. signature $(\mathscr{T}, \mathscr{C}, V, atr, F, mth)$, c is a logical variable, $C \in \mathscr{C}$:

$$\begin{array}{rlll}
F ::= & c & : \tau_C & \\
| & allInstances_C & : 2^{\tau_C}, \; C \in \mathscr{C} & \\
| & v(F) & : \tau_C \to \tau_\perp, & \text{if } v : \tau \in atr(C), \; \tau \in \mathscr{T} \\
| & v(F) & : \tau_C \to \tau_D, & \text{if } v : D_{0,1} \in atr(C) \\
| & v(F) & : \tau_C \to 2^{\tau_D}, & \text{if } v : D_* \in atr(C) \\
| & f(F_1, \dots, F_n) & : \tau_1 \times \dots \times \tau_n \to \tau, & \text{if } f : \tau_1 \times \dots \times \tau_n \to \tau \quad (*) \\
| & \forall\, c \in F_1 \bullet F_2 & : \tau_C \times 2^{\tau_C} \times \mathbb{B}_\perp \to \mathbb{B}_\perp &
\end{array}$$

- The formula above in prefix normal form: $\forall\, c \in allInstances_C \bullet \overset{f}{\neq} (\underbrace{x(c)}_{F_1}, \underbrace{27}_{F_2})$ $(*)$

Semantics

disjoint union

$\sigma : \mathscr{D}(\mathscr{C}) \nrightarrow (V \nrightarrow \mathscr{D}(\mathscr{T}) \cup \mathscr{D}(\mathscr{C}))$

- **Proto-OCL Types:**
  - $\mathcal{I}[\![\tau_C]\!] = \mathscr{D}(C) \,\dot\cup\, \{\bot\}$, $\quad \mathcal{I}[\![\tau_\bot]\!] = \mathscr{D}(\tau) \,\dot\cup\, \{\bot\}$, $\quad \mathcal{I}[\![2^{\tau_C}]\!] = \mathscr{D}(C_*) \,\dot\cup\, \{\bot\}$
  - $\mathcal{I}[\![\mathbb{B}_\bot]\!] = \{\textit{true}, \textit{false}\} \,\dot\cup\, \{\bot\}$, $\quad \mathcal{I}[\![\mathbb{Z}_\bot]\!] = \mathbb{Z} \,\dot\cup\, \{\bot\}$

- **Functions:**
  - We assume $f_\mathcal{I}$ given for each function symbol f (→ in a minute).

- **Proto-OCL Semantics** (interpretation function):
  - $\mathcal{I}[\![c]\!](\sigma, \beta) = \beta(c)$ (assuming β is a type-consistent valuation of the logical variables),
  - $\mathcal{I}[\![allInstances_C]\!](\sigma, \beta) = \mathrm{dom}(\sigma) \cap \mathscr{D}(C)$,

    i.e. if $\mathcal{I}[\![F]\!](\sigma,\beta)$ is alive in σ
  - $\mathcal{I}[\![v(F)]\!](\sigma, \beta) = \begin{cases} (\sigma\,(\mathcal{I}[\![F]\!](\sigma,\beta)))(v) & \text{, if } \mathcal{I}[\![F]\!](\sigma,\beta) \in \mathrm{dom}(\sigma) \\ \bot & \text{, otherwise} \end{cases}$ (if not $v : C_{0,1}$)
  - $\mathcal{I}[\![v(F)]\!](\sigma, \beta) = \begin{cases} \sigma(u')(v) & \text{, if } \mathcal{I}[\![F]\!](\sigma,\beta) = \{u'\} \subseteq \mathrm{dom}(\sigma) \\ \bot & \text{, otherwise} \end{cases}$ (if $v : C_{0,1}$)
  - $\mathcal{I}[\![f(F_1, \dots, F_n)]\!](\sigma, \beta) = f_\mathcal{I}(\mathcal{I}[\![F_1]\!](\sigma,\beta), \dots, \mathcal{I}[\![F_n]\!](\sigma,\beta))$,
  - $\mathcal{I}[\![\forall\, c \in F_1 \bullet F_2]\!](\sigma, \beta) = \begin{cases} \textit{true} & \text{, if } \mathcal{I}[\![F_2]\!](\sigma, \beta[c := u]) = \textit{true} \text{ for all } u \in \mathcal{I}[\![F_1]\!](\sigma,\beta) \\ \textit{false} & \text{, if } \mathcal{I}[\![F_2]\!](\sigma, \beta[c := u]) = \textit{false} \text{ for some } u \in \mathcal{I}[\![F_1]\!](\sigma,\beta) \\ \bot & \text{, otherwise} \end{cases}$

Semantics Cont'd

- Proto-OCL is a **three-valued** logic: a formula evaluates to *true*, *false*, or $\bot$.

- **Example**: $\wedge_\mathcal{I}(\cdot,\cdot) : \{\textit{true}, \textit{false}, \bot\} \times \{\textit{true}, \textit{false}, \bot\} \to \{\textit{true}, \textit{false}, \bot\}$ is defined as follows:

x_1	*true*	*true*	*true*	*false*	*false*	*false*	$\bot$	$\bot$	$\bot$
x_2	*true*	*false*	$\bot$	*true*	*false*	$\bot$	*true*	*false*	$\bot$
$\wedge_\mathcal{I}(x_1, x_2)$	*true*	*false*	$\bot$	*false*	*false*	*false*	$\bot$	*false*	$\bot$

  We assume common logical connectives $\neg, \wedge, \vee, \dots$ with canonical 3-valued interpretation.

- **Example**: $+_\mathcal{I}(\cdot,\cdot) : (\mathbb{Z} \,\dot\cup\, \{\bot\}) \times (\mathbb{Z} \,\dot\cup\, \{\bot\}) \to \mathbb{Z} \,\dot\cup\, \{\bot\}$

$$+_\mathcal{I}(x_1, x_2) = \begin{cases} x_1 + x_2 & \text{, if } x_1 \neq \bot \text{ and } x_2 \neq \bot \\ \bot & \text{, otherwise} \end{cases}$$

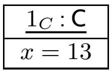
  We assume common arithmetic operations $-, /, *, \dots$
  and relation symbols $>, <, \leq, \dots$ with **monotone** 3-valued interpretation.

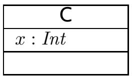
- And we assume the special unary function symbol $isUndefined$:

$$isUndefined_\mathcal{I}(x) = \begin{cases} \textit{true} & \text{, if } x = \bot, \\ \textit{false} & \text{, otherwise} \end{cases}$$

  $isUndefined_\mathcal{I}$ is **definite**: it never yields $\bot$.

Example: Evaluate Formula for System State

$\sigma :$ 

$\mathcal{S}:$ 

$$F = \forall\, c \in allInstances_C \bullet x(c) \neq 27$$

- Recall **prefix notation**: $\forall\, c \in allInstances_C \bullet \neq(x(c), 27)$

  Note: $\neq$ is a binary function symbol, 27 is a 0-ary function symbol.

- **Example**:

  $\mathcal{I}[\![\forall\, c \in allInstances_C \bullet \underline{\neq(x(c), 27)}]\!](\sigma, \emptyset) =$ *true*, because…

  $\mathcal{I}[\![\neq(x(c), 27)]\!](\sigma, \beta),\quad \underbrace{\beta := \emptyset[c := 1_C] = \{c \mapsto 1_C\}}$

  $= \neq_{\mathcal{I}}(\ \mathcal{I}[\![x(c)]\!](\sigma, \beta),\ \mathcal{I}[\![27]\!](\sigma, \beta)\ )$

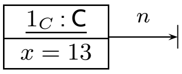
  $= \neq_{\mathcal{I}}(\underbrace{\sigma(\ \mathcal{I}[\![c]\!](\sigma, \beta)\ )}(x),\ 27_{\mathcal{I}}\ )$

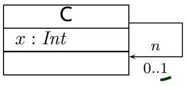
  $= \neq_{\mathcal{I}}(\ \sigma(\ \beta(c)\ )(x),\ 27_{\mathcal{I}}\ )$

  $= \neq_{\mathcal{I}}(\ \sigma(\ 1_C\ )(x),\ 27_{\mathcal{I}}\ )$

  $= \neq_{\mathcal{I}}(\ 13,\ 27\ ) =$ *true* …and 1_C is the only C-object in σ: $\mathcal{I}[\![allInstances_C]\!](\sigma, \emptyset) = \{1_C\}$.

More Interesting Example

$\sigma :$ 



$$\forall\, c\, \overset{\in allInstances_C}{\boxed{\cancel{\;\;}}} \bullet \underbrace{x(n(c)) \neq 27}_{\neq(x(n(c)), 27)}$$

- Similar to the previous slide, we need the value of

$\beta = \{c \mapsto 1_C\}$

$$\underbrace{\sigma\ (\ \underbrace{\sigma(\ \underbrace{\mathcal{I}[\![c]\!](\sigma, \beta)}_{\beta(c) = 1_C}\ )(n)}_{\sigma(1_C)(n) = \emptyset}\ )\ (x)}_{= \bot}$$

More Interesting Example

$$\forall c \overset{\in allInstances_C}{\cancel{:C}} \bullet x(n(c)) \neq 27$$

- Similar to the previous slide, we need the value of

$$\sigma\,(\,\sigma(\,\mathcal{I}[\![c]\!](\sigma,\beta)\,)(n)\,)\,(x)$$

  - $\mathcal{I}[\![c]\!](\sigma,\beta) = \beta(c) = 1_C$
  - $\sigma(\,\mathcal{I}[\![c]\!](\sigma,\beta)\,)(n) = \sigma(\,1_C\,)(n) = \emptyset$
  - $\sigma\,(\,\sigma(\,\mathcal{I}[\![c]\!](\sigma,\beta)\,)(n)\,)\,(x) = \bot$

by the following rule:

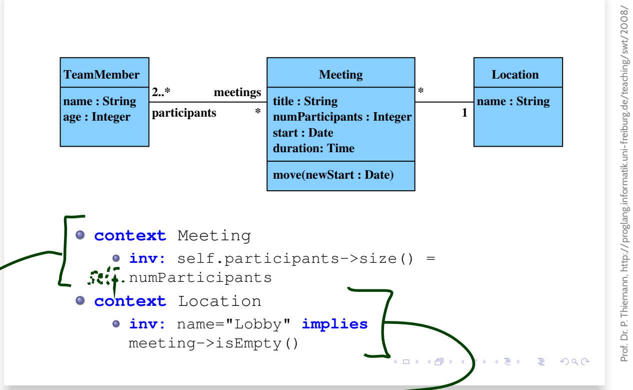
$$\mathcal{I}[\![v(F)]\!](\sigma,\beta) = \begin{cases} \sigma(u')(v) & \text{, if } \mathcal{I}[\![F]\!](\sigma,\beta) = \{u'\} \subseteq \mathrm{dom}(\sigma) \\ \bot & \text{, otherwise} \end{cases} \qquad \text{(if } v : C_{0,1}\text{)}$$

Object Constraint Language (OCL)

OCL is the same – just with less readable (?) syntax.

Literature: (OMG, 2006; Warmer and Kleppe, 1999).

Examples (from lecture "Softwaretechnik 2008")



- **context** Meeting
  - **inv:** self.participants->size() = self.numParticipants
- **context** Location
  - **inv:** name="Lobby" **implies** meeting->isEmpty()

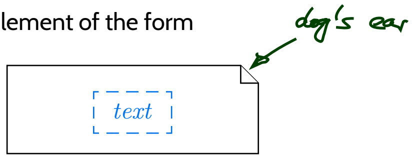
Prof. Dr. P. Thiemann, http://proglang.informatik.uni-freiburg.de/teaching/swt/2008/

$\forall\, self \in allInstances_{Meeting} \bullet size(participants(self)) = numParticipants(self)$

$\forall\, self \in allInstances_{Location} \bullet name(self) = \text{"Lobby"} \Rightarrow isEmpty(meeting(self))$

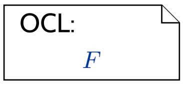
Where To Put OCL Constraints?

- **Notes**: A UML note is a diagram element of the form

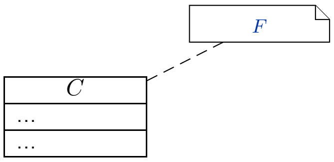


text can principally be everything, in particular comments and constraints.

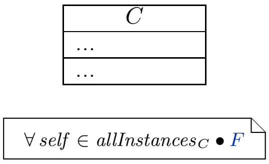
Sometimes, content is explicitly classified for clarity:



- Conventions:



stands for



Content

- **Class Diagrams**
  - concrete syntax,
  - abstract syntax,
  - class diagrams at work,
  - semantics: system states.
- **Object Diagrams**
  - concrete syntax,
  - dangling references,
  - partial vs. complete,
  - object diagrams at work.
- **Proto-OCL**
  - syntax,
  - semantics,
  - Proto-OCL vs. OCL.
- Putting it All Together:
  Proto-OCL vs. Software

Putting It All Together

Modelling Structure with Class Diagrams

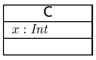
Definition. Software is a finite description S of a (possibly infinite) set $[\![S]\!]$ of (finite or infinite) computation paths of the form $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$ where

- $\sigma_i \in \Sigma$, $i \in \mathbb{N}_0$, is called state (or configuration), and
- $\alpha_i \in A$, $i \in \mathbb{N}_0$, is called action (or event).

The (possibly partial) function $[\![\cdot]\!] : S \mapsto [\![S]\!]$ is called interpretation of S.

- The set of **states** Σ could be the set of **system states** as defined by a class diagram, e.g.

$$\Sigma := \Sigma_{\mathscr{S}}^{\mathscr{D}} \qquad \mathscr{S} :$$



- A corresponding **computation path** of a software S could be

$$\boxed{\begin{array}{c}27_C : \mathsf{C}\\ \hline x = 0\end{array}} \xrightarrow{\tau} \boxed{\begin{array}{c}27_C : \mathsf{C}\\ \hline x = 1\end{array}} \xrightarrow{\tau} \boxed{\begin{array}{c}27_C : \mathsf{C}\\ \hline x = 3\end{array}} \xrightarrow{\tau} \boxed{\begin{array}{c}27_C : \mathsf{C}\\ \hline x = 4\end{array}} \xrightarrow{\tau} \cdots$$

- If a requirement is formalised by the Proto-OCL constraint

$$F = \forall\, c \in allInstances_C \bullet x(c) < 4$$

then S **does not** satisfy the requirement.

More General: Software vs. Proto-OCL

- Let $\mathscr{S}$ be an **object system signature** and $\mathscr{D}$ a **structure**.
- Let S be a **software** with
  - states $\Sigma \subseteq \Sigma_{\mathscr{S}}^{\mathscr{D}}$, and
  - **computation paths** $[\![S]\!]$.
- Let F be a Proto-OCL constraint over $\mathscr{S}$.
- We say $[\![S]\!]$ **satisfies** F, denoted by $[\![S]\!] \models F$, if and only if for all

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \in [\![S]\!]$$

and all $i \in \mathbb{N}_0$,

$$\mathcal{I}[\![F]\!](\sigma_i, \emptyset) = \textit{true}.$$

- We say $[\![S]\!]$ **does not satisfy** F, denoted by $[\![S]\!] \not\models F$, if and only if there exists $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \in [\![S]\!]$ and $i \in \mathbb{N}_0$, such that $\mathcal{I}[\![F]\!](\sigma_i, \emptyset) = \textit{false}$.
- **Note**: $\neg([\![S]\!] \not\models F)$ does not imply $[\![S]\!] \models F$.

Tell Them What You've Told Them…

- **Class Diagrams** can be used to **graphically**
  - visualise code,
  - define an **object system** ~~structure~~ Sig. $\mathscr{S}$.
- An **Object System** ~~Structure~~ Sig. $\mathscr{S}$ (together with a structure $\mathscr{D}$)
  - defines a set of **system states** $\Sigma_\mathscr{S}^\mathscr{D}$.
- A **System State** $\sigma \in \Sigma_\mathscr{S}^\mathscr{D}$
  - can be **visualised** by an **object diagram**.
- **Proto-OCL** constraints can be evaluated on **system states**.
- A **software** over $\Sigma_\mathscr{S}^\mathscr{D}$ satisfies a Proto-OCL constraint F if and only if F evaluates to *true* in all system states of all the software's computation paths.

References

References

Ambler, S. W. (2005). *The Elements of UML 2.0 Style*. Cambridge University Press.

Kopetz, H. (2011). What I learned from Brian. In Jones, C. B. et al., editors, *Dependable and Historic Computing*, volume 6875 of *LNCS*. Springer.

Lovins, A. B. and Lovins, L. H. (2001). *Brittle Power - Energy Strategy for National Security*. Rocky Mountain Institute.

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.

OMG (2006). Object Constraint Language, version 2.0. Technical Report formal/06-05-01.

Schumann, M., Steinke, J., Deck, A., and Westphal, B. (2008). Traceviewer technical documentation, version 1.0. Technical report, Carl von Ossietzky Universität Oldenburg und OFFIS.

Warmer, J. and Kleppe, A. (1999). *The Object Constraint Language*. Addison-Wesley.